GPU programming using C++ AMP

Petrika Manika
Dept. of Informatics
University of Tirana
petrika.manika@fshn.edu.al

Elda Xhumari
Dept. of Informatics
University of Tirana
elda.xhumari@fshn.edu.al

Julian Fejzaj
Dept. of Informatics
University of Tirana
julian.fejzaj@fshn.edu.al

Abstract

Nowadays, a challenge for programmers is to make their programs better. The word "better" means more simple, portable and much faster in execution. Heterogeneous computing is a new methodology in computer science field. GPGPU programming is a new and challenging technique which is used for solving problems with data parallel nature. In this paper we describe this new programming methodology with focus on GPU programming using C++ AMP language, and what kinds of problems are suitable for acceleration using these parallel techniques. Finally we describe the solution for a simple problem using C++ AMP and the advantages of this solution.

1. Introduction

The process of implementation of an algorithm as a solution for a difficult problem, requires a deep analysis. Although, today there are many tools that facilitate this work for the analysts and the process of translation into a programming language for the programmers. There are always difficulties when the execution speed is important. When the execution speed is not the main condition, then for programmers is easier and they can faster find a solution by building a source code, which contains instructions that are executed in series. When the primary condition of the proposed algorithm is the execution speed, then parallel programming becomes more important. Besides parallel source code, whose instructions are executed in parallel from CPU (Central Processing Unit), a new methodology is GPGPU programming.

General-purpose computing on graphics processing units (GPGPU, rarely GPGP or GP²U) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU)[1]. The architecture of graphics processing units (GPUs) is very well suited for data-parallel problems. They support extremely high throughput through many parallel processing units and very high memory bandwidth. For problems that match the GPU architecture well, it common to easily achieve a 2× speedup over a CPU implementation of the same problem, and tuned implementations can outperform the CPU by a factor of 10 to 100. Programming these processors, however, remains a challenge because the architecture differs so significantly from the CPU. This paper describes the benefits of GPU programming using C++ AMP language, and what kinds of problems are suitable for acceleration using these parallel techniques.

2. Performance Improvements

The world "Personal Computer" was introduced for the first time in 1975. Over the decades, the idea of having a personal computer become possible and real. Nowadays every person possesses various electronic machines from desktop computer, laptop up to smartphones. Over the years, the technology evolution made these electronic machines to work much faster. Manufacturers continued to increase the number of transistors on a single chip, but this faced with the problem of heat produced from this chips. Due to this problem, manufacturers started to produce multicore machines with two or more CPUs on a computer. However, adding CPU cores did not make everything faster.

We can divide softwares in two groups: parallel-aware and parallel-unaware. Parallel-unaware softwares use almost 1/4 or 1/8 of available CPU cores, while parallel-aware softwares can reach an execution speed 2x or 4x more than softwares of the second category, proportional to the numbers of CPU cores.

[1] General-purpose computing on graphics processing

2.1 Heterogeneous Platforms

In the last years, also the graphics cards have encountered a powerful development. A graphics processing unit (GPU) is a computer chip that performs rapid mathematical calculations, primarily for the purpose of rendering images[2]. GPU has a powerful parallel processing architecture, so it can render images more quickly than a CPU. GPU is a programmable and powerful computational device in its own right. The resulting performance improvements have made GPUs popular chips for other resource-intensive tasks unrelated to graphics.

GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate deep learning, analytics, and engineering applications. If CPU is the brain of the PC, GPU is called it's soul. Nowadays, we can find machines with two, four, seven CPU cores, but GPUs can have hundreds of cores. If we want to know the difference between a GPU and a CPU, let's see how they process tasks. A CPU consists of a few cores optimized for sequential serial processing, while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. Imagine a mix of GPU cores and CPU cores in a machine, in the same chip or not, this is a heterogeneous supercomputer.

In computing, FLOPS or flops (Floating-point Operations per Second) is a measure of computer performance, useful in fields of scientific calculations that make heavy use of floating-point calculations. For such cases it is a more accurate measure than the generic instructions[3]. So a 1 FLOP machine will do one "operation" in a second. Floating-point operations involve floating-point numbers and typically take longer to execute than simple binary integer operations. 1 Gigaflops has 1 billion FLOPS, and 1 Teraflops has 1000 Gigaflops. A typically CPU can achieve 100 GFLOPS. A typically GPU has 32 cores and has twice as many transistor as the CPU and can achieve 3000 GFLOPS.

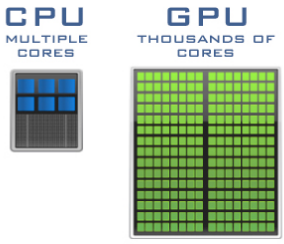

Figure 1: CPU vs GPU

The reason why GPU achieves this performance lies not in the number of transistors or the number of cores. Memory bandwidth is the rate at which data can be read from or stored into a semiconductor memory by a processor. Memory bandwidth is usually expressed in units of bytes/second[4]. The memory bandwidth of a CPU is roughly 20 GB/s, compared to the GPU's 150 GB/s. The CPU supports general code with multitasking, I/O, virtualization, deep execution pipelines, and random accesses. In contrast, the GPU is designed for graphics and data-parallel code with programmable and fixed function processors, shallow execution pipelines, and sequential accesses.

A strong point of GPUs is power consumption. A GPU can do 10 GFLOPS/watt, a CPU can do 1 GIGAFLOPS/watt. The battery life of a machine is very important, especially in handheld devices. In most cases, users prefer not to use applications that consume battery fast, replacing them with similar applications that do not consume the battery. If we study the memory accessed from this chips, CPU has a large cache for the data that it access, in order to not wait for the execution of the processes that read data from primary or secondary memory, since the CPU use often the same data. GPUs have smaller caches, but use a massive number of threads and some threads are always in a position to do work. GPUs can prefetch data to hide memory latency. Different from CPU, a GPU have small cache, because the probability to access the same data more than once is small.

Nowadays we can find a lot of CPU programming languages. C++ is a popular CPU programming language. It is a main language of choice when it comes to power and performance. Choices are few when it comes to general-purpose GPU programming (GPGPU). Developers need a way to increase the speed of their applications or to reduce the power consumption of a particular calculation. A solution is

[2] http://searchvirtualdesktop.techtarget.com/definition /GPU-graphics-processing-unit

[3] FLOPS - https://en.wikipedia.org

[4] FLOPS - https://en.wikipedia.org

using heterogeneous computation with GPUs, in association with CPUs. One bad side of this choice is the restriction on the nature of softwares that can be built using this methodology.

2.2 GPU Architecture

GPUs have shallow execution pipelines, small cache, and a massive number of threads performing sequential accesses. Threads are arranged in groups. This groups are called warps.

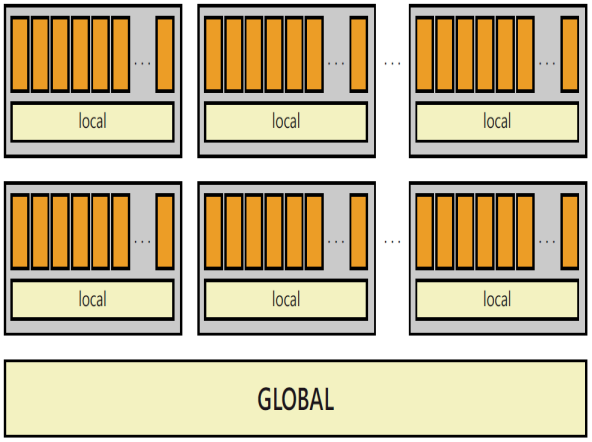

Figure 2: GPU Architecture

Warps run together and can share memory and cooperate. A powerful ability is that GPUs can switch these groups of threads extremely fast, so if a group of threads is blocked, another group of threads executes. When adjacent threads use adjacent memory locations, the way the memory is read provides good speed performance. The bad side is when threads in a group are accessing memory that is not near the memory being accessed by other threads in that group, performance will suffer.

Developers of high level programming languages do not have the necessary to now the architecture of CPUs as long as there are tools like compilers. If a developer wants to write code that will be executed on GPUs, he should know basics from GPU architecture.

2.3 Performance improvement through parallelism

CPU works with both parallel code with parallel data and sequential code. GPU works best on problems that are data-parallel. There are problems that can easily be divided into sub-problems which can be executed in parallel and independent of each other. But there are also problems that if you treat them in a manner, you will not be able to divide them into units that can be executed in parallel, but they must be treated in another way, so you will be able to divide them into independent units that can be executed in parallel. The conclusion is that when programmers think about the solution (source code) of a problem and the execution speed is the priority, they must think about a parallel solution. They need to design the algorithm differently to create work that can be splited across independent threads.

For example, a problem with a parallel nature is the addition of two matrices. If we want a quick and simple solution, in C++ it would be as follows:

```
int       M1[n][n],M2[n][n],M_sum[n][n];
for(int   i=0;i<n;i++)
{
  for(int j=0;j<n;j++)
    M_sum[i][j]=M1[i][j]+M2[i][j]
}
```

If we have two matrices 100x100, with 10000 integer numbers, the above code will take 10000 additions, 10000 operations that will be executed one by one. If we want e fast execution, we can think about a parallel solution, so we can split the operations among 10000 threads and all the additions could be done at once.

Another example is the problem of finding the highest value in a vector (array, collection). The fast solution is to traverse the array one element at a time and comparing each element to a maximum value (represent the currently highest value), then updating the maximum value with the current element of the array, if it is larger. If the array has 100000 elements, this will take 100000 comparisons. If the priority is the fast execution of the algorithm, we can think for a parallel solution, so we can select 1000 threads and they can take on 1000 items each. After the calculations, each thread will select the highest value of its portion of the array. That way you could evaluate every number in the time it takes to do just 1000 comparisons. After that, a 1001st thread could compare all the results from the threads for finding the highest

value. Problems that involve large quantities of data are candidates for parallel processing. Some fields where we can find this kind of problems are:

- Real-time control systems
- Scientific modeling and simulation
- Gaming
- Financial Simulation
- Image processing

One way to reduce the amount of time spent in the sequential portion of your application is to make it less sequential—to redesign the application to take advantage of CPU parallelism as well as GPU parallelism**.**

3. GPU Programming frameworks

Nowadays exists some GPU programming languages that developers can use to build parallel softwares. These parallel programming languages have their advantages and disadvantages. Some of these platforms are:

3.1 OpenCL

OpenCL is the dominant open general-purpose GPU computing language. OpenCL is supported on Intel, AMD, Nvidia and ARM platforms. It is a framework for writing programs that execute across heterogeneous platforms.

3.2 CUDA

CUDA is a parallel computing platform and application programming interface (API) model created by NVidia. The CUDA platform can work with C, C++, and FORTRAN programming languages.

3.3 C++ AMP

C++ AMP (C++ Accelerated Massive Parallelism) [Mil1] accelerates the execution of C++ code by taking advantage of the data-parallel hardware that's commonly present as a graphics processing unit (GPU) on a discrete graphics card[5]. C++ Accelerated Massive Parallelism (C++ AMP) is a native programming model that contains elements that span the C++ programming language and its runtime library. C++ AMP is a library implemented on DirectX 11 and an open specification from Microsoft for implementing data parallelism directly in C++. This language is easier to use and contains many libraries for building data-parallel applications.

4. A C++ AMP Solution

To show more clearly how to use C++ AMP for solving a data-parallel problem[6], I will present the following example. The problem is simple and has a parallel nature, matrix multiplication.
Let's take a mathematical or financial application where part of a process is the multiplication of matrices, but possible scenarios are multiplication of matrices with small sizes and the multiplication operation will not be executed many times, then a serial source code for this operation would not take a long execution time and the parallel source code would be excessive. But, imagine a scenario with 200 matrices with 40000 elements each. From here we have 100 multiplication operations and the serial source code would take a long execution time, while if 100 operations would be executed in parallel, utilizing the facilities that GPU provides, the execution time would be smaller.
Below is a simple un-parallel function in C++ for the multiplication of two matrices:

```
multiplication(vector<vector<int>>& T1,
vector<vector<int>>& T2, vector<vector<int>>&
T3, const int n, const int m, const int k)
{
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<k; j++)
        {
            int sum = 0;
            for(int z=0; z<m; z++)
```

[5] https://msdn.microsoft.com/en-us/library/hh265137.aspx

[6] C++ AMP Overview - https://msdn.microsoft.com/en-us/library/hh265136(v=vs.120).aspx, retrieved November 3, 2015.

```
            sum += T1[i][z]*T2[z][j];
        T3[i][j] = sum;
      }
    }
}
```

If the size of the matrices T1 and T2 would be 200x200, the execution time of this function in a moderate computer would be 3.36 sec. If we take the scenario explained above, thus 100 multiplication operations, the execution time would be 336 sec, or 5.6 minutes.
Let's see the C++ AMP parallel function for the multiplication of two matrices:

```
multiplication_parallel(vector<vector<int>>& T1,
vector<vector<int>>& T2, vector<vector<int>>&
T3, const int n, const int m, const int k)
array_view<const int, 2> a(n, m, T1), b(m, k, T2);
array_view<int, 2> c(n, k, T3);
c.discard_data();
parallel_for_each(c.extent, [=](index<2> idx)
restrict(amp)
{
  int row = idx[0]; int col = idx[1];
  int sum = 0;
  for(int i = 0; i < b.extent[0]; i++)
    sum += a(row, i) * b(i, col);
  c[idx] = sum;
});
c.synchronize();
```

This version use the array_view data structure of C++ AMP library. The parallel_for_each function is the main function that does all the parallel job from the computer's GPU [Gas2]. This function operates over an extent—the shape of the extent is what controls the number of threads that do the work. In a moderate computer, the execution time is 20x faster than the un-parallel solution. The difference in execution time between the first function and the second is obvious.

5. Conclusions

If the primary condition of the proposed algorithm for a problem is the execution speed, then parallel programming becomes important. GPGPU programming is a new and challenging technique which is used for solving problems with data parallel nature. Some fields where we can find this kind of problems are: real-time control systems, scientific modeling and simulation, gaming, financial simulation, image processing, etc. For problems that match the GPU architecture well, it is common to easily achieve a 2× speedup or more over a CPU implementation of the same problem, and tuned implementations can outperform the CPU by a factor of 10 to 100. GPU is a programmable and powerful computational device in its own right. GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. The memory bandwidth of a CPU is roughly 20 GB/s, compared to the GPU's 150 GB/s. A strong point of GPUs is power consumption. A GPU can do 10 GFLOPS/watt, a CPU can do 1 GIGAFLOPS/watt.
If we want to build code that will be executed in parallel, GPGPU is a good candidate as a new technology and C ++ AMP is a language that provides facilities while programming and necessary libraries.

References

[Mil1] Gregory, Kate, and Ade Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®.* " O'Reilly Media, Inc.", 2012

[Gas2] B. R. Gaster and L. Howes, "*Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck,*" Computer, vol. 45, no. 8, pp. 42–52, Aug. 2012